# Research report 140

## COMPUTER SYSTEM DEPENDABILITY: AN INTRODUCTION

**Andrew M Lord**

(RR140)

More reliance is being placed on computer systems in many applications. Such systems need to be dependable; we must be able to depend on them to provide the required level of service without failure. This report gives an introduction to the idea of dependability and gives and overview of some of the techniques used in making computer systems dependable. In particular, specific fault-tolerance techniques for making various system components more dependable are reviewed. Annotated references are given for work done in many of the areas in an Appendix.

---

This work was supported by research grant GR/E11521 from the Science and Engineering Research Council.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

April 1989

# Contents

# 1 Introduction

No computer system can be guaranteed not to fail, yet computers are being used in some applications where a failure may be fatal (such as controlling an aircraft or the space shuttle) and in others where a total breakdown needs to be avoided at all costs (such as telephone exchanges and air-traffic control). We need to ensure that systems are *reliable* (i.e. that the chance of them failing is sufficiently low) and *available* (i.e. that the time the systems are not working is small). Together these give what we mean by a *dependable* system. Making systems dependable is not a new problem – ever since the times of vacuum tubes and CRT storage the problem of making systems more reliable and available has been considered. The problem has been recently highlighted by press "horror stories" of systems failing (see Borning, 1987, and Thomas, 1988, for collections of such examples).

The initial solution (taken from the 1950's) was to try and avoid faults by ensuring that the system components were as reliable as possible (termed *fault–avoidance*). This makes the systems more reliable, but there is still a problem if the computer eventually fails. This has led to the development of *fault–tolerance* techniques whereby faults may manifest themselves, but they are tolerated and the system continues to operate and to meet its specification. Fault–tolerance is achieved through the use of *redundancy* in hardware, software and time. Different redundancy techniques are useful for different components in a system. For example, circuits can be replicated and then operated in parallel, CRC codes may be added to messages, databases may be replicated at different sites, multiple copies of a process may execute with the majority result taken, and duplicate software written by different people can take over if the original software fails. We must remember, though, that there is always a limit to the dependability of a system – there is always the problem of how faults in the fault detection/correction mechanism can be detected.

Developing a dependable system involves applying such techniques to appropriate parts of the system design. Once developed it is then required to measure the dependability to ensure it meets the requirements, which may lead to a redesigning of the system using different techniques if these are not met. Many methods have been devised for testing the dependability of systems, particularly at the circuit level. Also, many models have been developed for evaluating the reliability of a system under a given set of fault assumptions. This *validation* of a system goes hand–in–hand with its design. Also of use here are formal methods which have been applied to prove various dependable protocols and algorithms correct. Despite being quite rigorous the design of dependable systems has traditionally had a rather *ad hoc* feel, though more recently models for distributed systems have been developed which include dependability techniques as an integral part. These designs will probably become the standard.

This report presents an introduction to the work done in the area of dependability together with a set of annotated references. The work is viewed within the framework of concepts proposed by the IFIP and IEEE fault–tolerance groups (see Laprie, 1985). Dependability impairments, means and measures are discussed in general terms. Then specific methods for achieving dependability are examined, focusing on different aspects of a computer system (such as communications, processes and data). A set of annotated references to each of the areas is given in the Appendix, and a complete set of references is given at the end of the report. An additional list of references which are not cited in the report, but may be useful, is also given.

# 2 Dependability

So far the use of terms has been intuitive. This is now rectified with definitions for the common terms, based on those given by Laprie (1985). More details of these terms, together with other related terms are given in later sections. A useful graphical guide to the terms is given in Figure 1.

A given computer *system* delivers a specified *service* to its users. The specification describes the systems functionality, performance requirements, and so on. The *dependability* of such a system is "the quality of the delivered service such that reliance can be justifiably be placed on this service". The notion of *reliance* here is hard to define, being based on the users expectations of the system's service, which may need to take into account psychological factors and also the experience of the individual. For a system to be dependable we need to be able to justify this reliance in some way. The term 'system' has a wide application and we must take care not to limit its use. For example, a physical computer (a system) may be composed of CPU, memory and data bus systems. In turn the CPU may be made up of various sub-systems (e.g. ALU, registers). The notion of dependability can be applied recursively to each of these systems.

A system's dependability is impaired by *failures* which occur when the system deviates from its specified

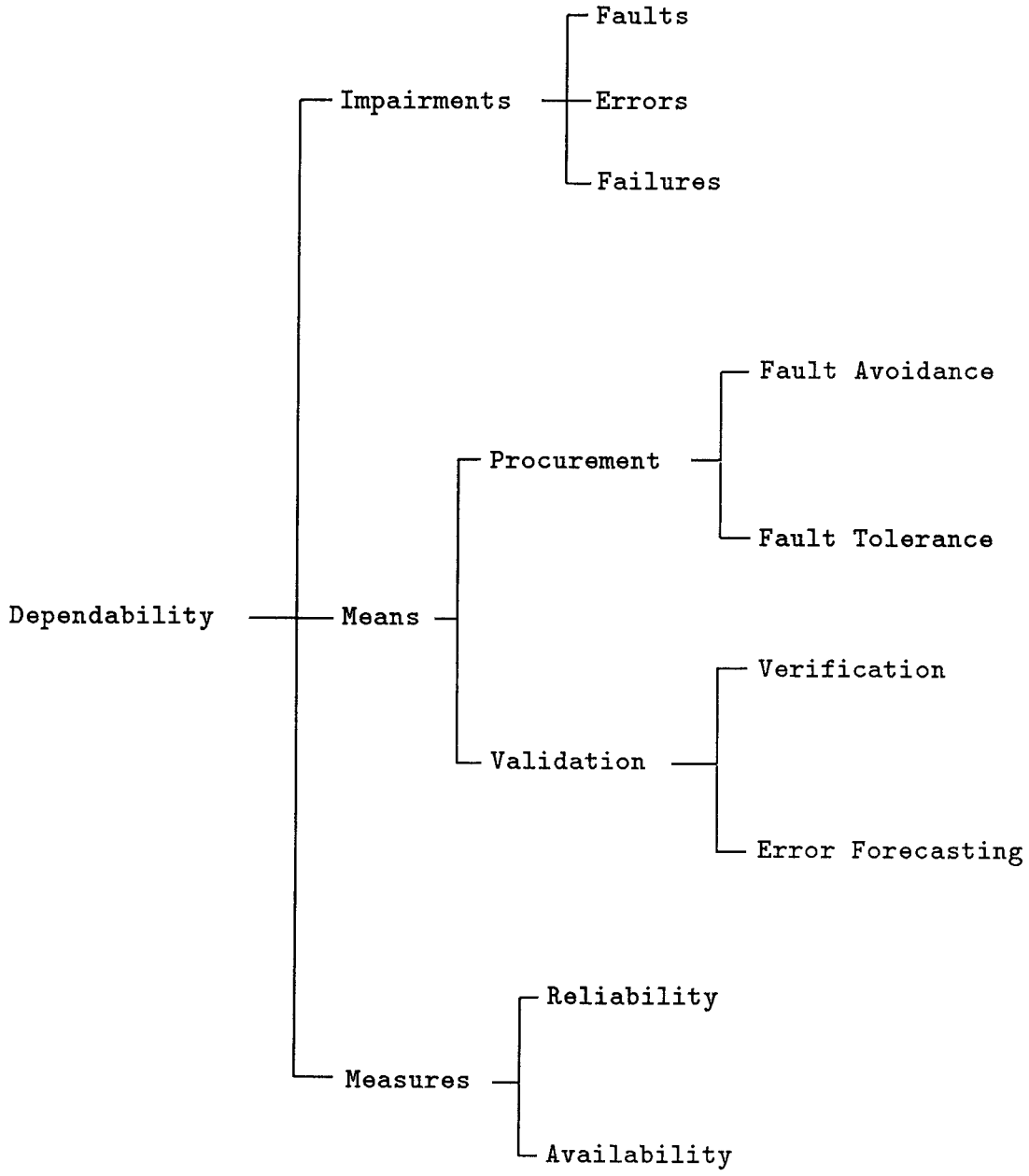


Figure 1: Dependability Terminology (taken from Laprie, 1985)

service. A failure can be traced to an *error* in part of the system state. The cause of an error is a *fault*. For example, within a memory system a physical fault may cause an AND gate to always output the value 1 regardless of inputs; this may lead to a parity error which is viewed by the user of the memory (e.g. a CPU) as a memory failure.

The means of achieving system dependability are by the combined application of a set of methods which can be classed into fault–avoidance, fault–tolerance, verification and error–forecasting methods. *Fault–avoidance* methods give means for preventing, by construction, fault occurrence. *Fault–tolerance* methods provide means of achieving the system specification despite the presence of faults, by the use of *redundancy*. *Verification* methods (ill termed *error–removal* methods by Laprie) are used to show that the system will meet its specification even in the presence of faults. *Error–forecasting* methods provide the means to estimate, by evaluation, the presence, creation and consequences of errors. Fault–avoidance and fault–tolerance constitute *dependability procurement*: how to provide the system with the ability to deliver the specified service. Verification and error–forecasting constitute *dependability validation*: how to reach confidence in the system's ability to deliver the specified service. Thus in developing a particular system, fault–avoidance methods may not be sufficient to prevent fault occurring, and hence some fault–tolerance methods are used. Verification and error–forecasting methods are used to see if the system meets then its specification. If not then new fault–avoidance or fault–tolerance methods are introduced. These in turn are verified, and so on until the system is dependable.

Dependability is measured in terms of reliability and availability. *Reliability* is a measure of the time to failure from an initial reference point. *Availability* is a measure of the proportion of time the system will meet its specification. It is possible for a system to have a high availability but low reliability. For instance, a system that fails frequently will be inherently unreliable, but if it can be restarted quickly enough it may have a high availability.

Dependability is a relatively new term, replacing the previously used term *reliability*. The reason was to remove the confusion between the general meaning of reliability (reliable systems) from reliability as a mathematical quantity (system reliability). Also, it shows how notions of reliability, maintainability, safety etc. are all measures corresponding to distinct perceptions of the same attribute. The other terms defined have been in common use for much longer with similar definitions, and they have just been brought together into the dependability framework. Unfortunately, not everyone uses the same terms: in particular, the difference between faults, errors and failures is sometimes not made. However, the terms used here are widely accepted.

**Useful References:** Laprie (1985), Anderson and Lee (1981) (1982), Randell, Lee and Treleaven (1978), Avižienis and Kelly (1984)

# 3 Impairments to Dependability

Making a system dependable involves overcoming faults which impair the systems dependability. A fault creates a *latent error* which becomes *effective* when activated. When the error affects the delivered service a *failure* occurs. These terms can be widely applied, not just to hardware situations: for example, a programmers mistake is a fault (commonly termed *bug*), the consequence being a (latent) error. This error will only become effective when a certain situation arises in the program execution. When this effective error produces erroneous data which affects the delivered service a failure occurs. Also, an incorrect entry of data by an operator is a fault; the resultant altered data being an error, etc. We can classify these faults into physical and human–made faults as follows:

**physical faults:** adverse physical phenomena, either internal (physico–chemical disorders: threshold changes, short–circuits, open circuits, ...) or external (environmental perturbations: electro–magnetic perturbations, temperature, vibration, ...).

**human–made faults:** imperfections which may be:

- **design faults,** committed either a) during the system initial design (broadly speaking, from requirement specifications to implementation) or during subsequent modifications, or b) during the establishment of operating or maintenance procedures.
- **interaction faults:** inadvertent or deliberate violations of operating or maintenance procedures. Deliberate violations include "undesired accesses" in the sense of computer security and privacy.

[taken from Laprie (1985)]

Physical (sometimes termed *operational*) faults can be classified by their duration, extent and value (see Avižienis, 1976). Design faults need not just occur in software; a hardware design may also have design faults in it.

When applying these terms to a system we must remember that a system may be composed of various components, each of which is another system. Thus a fault in one system may be viewed as a failure in one of its components. At the lowest level of atomic systems there will be "elementary" failures.

**Useful References:** Laprie (1985), Avižienis (1976)

# 4 Measures of Dependability

The two main measures of dependability are reliability and availability, as mentioned. *Reliability* is a measure of the time to failure from an initial reference point, or equivalently a measure of the degree of tolerance against faults or of the ability of a system to meet its specification. *Availability* is a measure of the proportion of time the system will meet its specification, and embodies the notion of failure frequency. Three common dependability measures are Mean Time To Failure (MTTF), Mean Time Between Failures (MTBF), and Mean Time to Recovery (MTTR). Often these measures are given as probabilities, usually based on mathematical models of the system. There is a link between these measures and those for error forecasting outlined in Section 6.2, the interest here being in failures rather than the lower–level errors. See Stiffler (1986) and McConnel and Siewiorek (1982) for introductions to reliability estimation.

Another measure often given for a system is *maintainability* which can be viewed as a measure of the time that the system specification is not met, or of the time to service restoration. This can be deduced from the system availability.

An important concern is the consequences of a failure on its environment. We can identify different magnitudes (or *modes*) of failure. In particular the notion of *benign* and *malign* (or *catastrophic*) failures is useful. The consequences of benign failures are of the same order of magnitude (usually in terms of cost) as those of the service delivered in the absence of failures, whereas catastrophic failures have much greater consequences. Using these notions to generalise reliability gives us a measure of the time to catastrophic failure, i.e. *safety*. It is possible for a reliable system (in terms of benign failures) to be unsafe. For example, an automatic braking system may bring a vehicle to a stop within the specified time but may cause an accident in doing so. See Leveson (1986) for a detailed overview of software safety issues.

**Useful References:** Stiffler (1986), McConnel and Siewiorek (1982)

# 5 Dependability Procurement

Dependability is obtained (procured) by fault–avoidance and fault–tolerance. Fault–avoidance involves the *a priori* elimination of faults. In practice it hasn't been possible to assure this elimination of faults. Hence redundancy in hardware, software and time is introduced into a system to make it dependable, by the so called fault–tolerance methods. Fault–avoidance is briefly looked at below, but the main emphasis in this section is on a detailed overview of fault–tolerance. Some such methods used for different components within a distributed computer system are reviewed in Section 7.

## 5.1 Fault Avoidance

Fault–avoidance is achieved by obtaining the most reliable components within the cost and performance constraints given in the specification; by use of thoroughly refined techniques for the interconnection of components and assembly of subsystems; by packaging the hardware to screen out expected forms of interference; and by carrying out comprehensive testing to eliminate hardware and software design faults (Avižienis, 1976). Fault avoidance, as mentioned, is not sufficient to prevent system failure and so needs supplementing by manual maintenance procedures to return the system to an operating condition after failure. To help with maintenance, some built–in error-detection, diagnosis and retry techniques are often provided. Fault–avoidance is the traditional approach to achieving system dependability, but in certain situations has been found unacceptable. In particular because of the unacceptable delays incurred in real–time systems (e.g. aircraft control), the inaccessibility of some systems to maintenance (e.g. space satellites), and the

excessively high costs of lost time and of maintenance in many installations. Hence the development of fault–tolerance techniques.

**Useful References:** Avižienis (1976), McCluskey (1986)

## 5.2 Fault–Tolerance

Fault–tolerance alleviates many of the shortcomings of fault–avoidance. Faults are expected to be present and to induce errors, but the ability of errors to cause failures is counteracted by the use of redundancy. Thus fault–tolerance is performed by *error processing* which may be automatic or operator–assisted. Two phases can be identified: *effective error processing* aimed at bringing the effective error back to a latent state, and *latent error processing* aimed at ensuring that the latent error does not become active again. Effective error processing may take the form of *error recovery* or *error compensation*, which are discussed below. The actual working out of these methods is strongly linked with the design and characteristics of the particular system being made dependable: they simply provide a useful framework for approaching the problem.

### 5.2.1 Error Recovery

Error recovery involves substituting an error–free state for the erroneous state in the system. Error recovery can take the form of bringing the system state back to a state occupied prior to the error become effective (*backwards error recovery*), or of finding a new state from which execution can continue without failure (*forwards error recovery*). Both forms require *error detection* and *damage confinement and assessment* (Anderson and Lee, 1981), as explained below.

In order to recover from a fault in a system its effects must first be detected. Ideally, error detection is based on checks that the system specification is bring met, and these checks are independent of the system itself. For example, there may be component checks testing memory CRC codes, timing checks to see if real–time constraints are met, or coding checks which test the consistency of the values of a set of variables. This idea can be generalised into the notion of standard and exceptional domains, with an error detected by testing to see if the system state is in the exceptional domain. In principle the more error detection that is performed, the more dependable the system since if all errors were detected and appropriate fault–tolerance techniques applied, then no no fault could lead to system failure. In practice there are limitations due to the ability to test the specification and the run–time overheads of checking.

There is likely to be a delay between the manifestation of a fault and the detection of the resulting errors. It is possible that in this delay invalid information might have spread within the system, causing more errors. Thus there is a need to assess the extent to which the system state has been damaged, prior to attempting recovery. This assessment will depend on the system structure, and decisions need to be made in system design as how to confine the damage a fault may induce. A useful structuring concept in this context is that of *atomic actions* which group the activities of a set of system components such that there are no interactions between that group and the rest of the system for the duration of the activity. The widely used notion of *transactions* within database systems is an example of this concept (Kohler, 1981). Atomic actions thus bound the effects of faults that occur within them giving natural confinement and assessment.

Once these two passive phases have been achieved, the active phase of changing the system state can be performed. Backward error recovery depends on the provision of *recovery points* at which the state of the system is recorded, and can be later reinstated to. Two common techniques for obtaining recovery points are checkpointing (Russell, 1980) and audit trails (Verhofstad, 1978). System structures such as recovery blocks (Randell, 1975) and recoverable atomic actions are based on backward error recovery. Such recovery is not so dependant on damage assessment and is capable of providing recovery from arbitrary faults. Forwards error recovery, on the other hand, is much more dependant on having identified the fault and its effects in order to construct a new state from which system execution can continue. Many common techniques for forwards error recovery are based on exceptions (Best and Cristian, 1981; Anderson and Lee, 1981). Such recovery is inappropriate for tolerating unanticipated faults, but can be more efficient than backwards error recovery for specific faults. Forwards and backwards error recovery are not of course exclusive, for example backward recovery may be attempted first, and if the effective error persists then forward recovery may be attempted.

### 5.2.2 Error Compensation

For error compensation, systems are designed with enough redundancy so that they can deliver an error–free service even with an erroneous internal state. A classic example is that of state machine systems (Schneider, 1986) where multiple copies of a system component operate in parallel with the output of the resulting component being the majority of the outputs of the copies. Error detection is not required since errors are masked. However, errors typically lead to a reduction in the redundancy available to tolerate any further faults, and so in practical systems error detection is performed in order to decide if any latent error processing should be done to restore the redundancy. Error compensation has some implicit notion of damage confinement; for example, in state machine systems damage is confined to component copies. Error compensation techniques have the advantage over error recovery techniques in that the time overhead is much lower, and so error compensation is perhaps essential for real–time systems. This is, of course, at a cost of increased structural redundancy.

### 5.2.3 Latent Error Processing

If effective error processing has removed the error, or makes its likelihood of recurring low enough then latent error processing is not undertaken. However, this is not always the case, as in the example of state machine systems where a (latent) internal error still exists despite compensation. Here latent error processing is undertaken to reconfigure the system so that the processor in error is removed or replaced by a repaired processor (see Schneider, 1986).

**Useful References:** Anderson and Lee (1981), Randell, Lee and Treleaven (1978), Laprie (1985), Schneider (1986)

# 6 Dependability Validation

Dependability validation provides a means to reach confidence in the systems ability to deliver the specified service. There are two sets of such means: verification methods and error forecasting methods. These are typically utilized throughout the life cycle of system development. Verification methods include the use of testing and formal methods. Error forecasting methods are based on mathematical models of the system and on the tested properties of its components; they give an estimate of the presence, creation and consequences of errors. Different validation methods have been developed, such as those by the U.S. Department of Defence outlined by Borning (1987), and those of NASA outlined by Czeck, Siewiorek and Segall (1985). These provide a rigorous framework for validating a system, ensuring that no part of the system is overlooked.

## 6.1 Verification

Given the safety–critical nature of many computer systems, it is very important that we have as much confidence as possible that they will meet the required specifications. Because of this there has been an increased interest in the use of formal methods in this process, in particular in the development of proof systems within which it is possible to *prove* that the specifications will be met for a given set of fault assumptions. Proofs give the possibility of achieving absolute confidence in the system, without room for error. However, so far such an ideal is not within reach since the construction of proof systems, and the actual proofs are far from trivial for complete computer systems. Nevertheless, formal methods have been applied to the verification of different parts of systems. Much effort has been put into proving that communication protocols and agreement protocols are correct and it is usual for such proofs to be given along-side the presentation of the protocols. For example, see Koomen (1985) who applies CCS to communication protocols, and Dolev and Strong (1983) who present an agreement algorithm with its associated proof. Proof systems have been developed for languages which have special fault–tolerance constructs in order to show that programs written in them will meet their specifications; for example Joseph, Moitra and Soundararajan (1984) and Schlichting (1982). The recently released VIPER processor has been designed to be used in dependable applications, and a proof that it met its specification was made (see Pountain, 1988, and Cohn, 1987). It appears that the use of such techniques will increase, as they are being more commercially accepted.

The more traditional method of verification is that of testing. The use of testing can only show the absence of errors, but not their presence unlike the use of formal methods. However, sufficiently rigorous

testing can give a good enough level of confidence in the system being tested. To take an example, the FTMP multi-processor designed for aircraft applications has been tested, with experiments including some to measure the response time to interrupts and exceptions, the execution time of high–level language instructions, to transfer blocks of data to and from memory, and the overhead and variation in fault detection and system configuration software (see Czeck, Siewiorek and Segall, 1985).

## 6.2 Error Forecasting

The terms failure, error and fault can be applied at different levels in a system, with a failure at one level being an error at some higher level. Hence methods for finding the reliability of a system can be applied equally to error forecasting, only with the terms suitably adjusted in meaning. Error forecasting methods are usefully applied once a system, or a component, has been designed, and in parallel with verification techniques. These methods, as mentioned, are based upon mathematical models of the system which cannot give a precise representation of the system, but are good enough for useful estimates to be made. However, many of the underlying assumptions and the applicability of these models does continue to be questioned. Measurements of hardware reliability were made first, and current approaches to measuring software reliability basically parallel those used for hardware reliability, with appropriate modifications to take into account the difference between hardware and software situations. Goel (1985) identifies four different classes of models: times between failure (TBF) models, failure count models, fault seeding models, and input domain based models. Each of these uses known details about failures, such as their frequency and extent. Assumptions have to be made about the system, for example TBF models assume independent times between faults and equal probability of the detection of each fault. Also, models have to be carefully chosen, and goodness-of-fit test made to check that the model is good enough for the given system. Computer–aided packages for estimating reliability have been constructed, such as the CARE-III system (Stiffler, 1986), and these use more detailed models giving more accurate or useful estimates, though at the expense of more details needing to be supplied.

**Useful References:** Czeck, Siewiorek and Segall (1985), Pountain (1988), Schlichting (1982), Goel (1985), Stiffler (1986)

# 7 Dependable Systems

Having introduced the notion of dependability and outlined in broad terms how to go about achieving it, let us look at some more specific examples. In this section methods for making six components of distributed computing systems dependable are examined: those for circuits, networks, data, processes and software. These components do have a significant amount of overlap, yet it is useful to approach each separately. In addition the problem of reaching agreement between a set of processes, utilized by some fault–tolerance methods, is looked at, including algorithms for clock synchronization.

## 7.1 Dependable Circuits

A widely used redundancy technique used for making circuits dependable is that of *coding*, for which an extensive theory has been constructed since the late 1960's. The basic idea is that of adding (redundant) codes to circuits, such as parity codes, which are used to detect and possibly correct errors. Example codes used in memory and tape systems, and for testing the results of arithmetic and logical operations are given by Tohma (1986) and Bose and Metzner (1986). Utilization of error–detecting codes does not necessarily guarantee the detection of a fault developed in a circuit: for example there may be a fault in the code checker. Hence there is a need for self–checking checkers, and self–checking circuits in general. It is also important to design circuits so that they can be easily tested, and faults easily detected; see McCluskey (1986).

**Useful References:** Pradhan (1986a)

## 7.2 Dependable Networks

With the trend towards distributed computing, making communications dependable is of increasing importance. The communication medium is more prone to error than most other components in a system, yet if a variable time delay can be tolerated then virtually error–free communication can be assured by

fairly straight–forward protocols. Such protocols for communication between two stations are based on the following three principles:

1. provide sufficient error checking (by means of error coding, e.g. CRC codes) to make negligible the chance of undetected error.
2. provide for retransmission of any data that are not positively acknowledged.
3. provide sequence number identification for each message unit to allow recognition of unrequested retransmission and unambiguous reassembly of messages received out of order.

These protocols have been available since the 1960's; see Bose and Metzner (1986) of an overview of some of these.

The problem of the time taken to achieve dependable communication is important. In the ISIS distributed system (Birman, 1986) this latency was found to be a major factor limiting performance, and led to the development of a set of communication primitives, each providing a certain type of communication with different timing requirements (see Birman and Joseph, 1986).

In networks with more than two stations the problem of constructing dependable routing algorithms appears. Routing algorithms must be able to tolerate host, interface message processor, and line failures, and be able to cope with changes of topology. Routing algorithms can be grouped into adaptive algorithms (which base their routing decisions on measurement or estimates of current traffic and topology) and nonadaptive algorithms (which don't). See Tanenbaum (1981, Section 5.2) for further details of the problems involved.

One particular kind of communication that is common to most networks is that of *broadcasts*, whereby a message is sent to all stations in a network. An introduction to broadcasting protocols in radio networks is given by Chlamtac and Kutten (1985) together with bounds on the delay for message broadcast. Drummond (1986) characterises the time complexity of broadcasts with respect to communication models, in addition to presenting broadcast protocols for networks of different degree (from complete to point–to–point networks). Farley (1987) describes network configurations that incur the minimum possible delay for broadcasting, given a simple fault model.

Another common (slightly higher–level) communication mechanism is that of Remote Procedure Calls (RPCs). RPCs package communication protocols such as those discussed and provide programmers with a communication mechanism which naturally extends the idea of procedures. They have been used in many operating systems, such as the Cambridge Distributed Computing System (see Bacon and Hamilton, 1987). The semantics of RPCs have been classified by Nelson into "exactly once" semantics, where if a client's call succeeds then exactly one execution has taken place at the server, and "at least once" semantics, where if a client's call succeeds then at least one execution has taken place at the server. The Newcastle RPC mechanism (Shrivastava and Panzieri, 1982) uses exactly once semantics, with the main dependability feature at the RPC level being the ability to discard unwanted messages, and any other dependability features requires for making applications dependable included at a higher–level dealing with atomic actions. Another implementation of RPCs is given by Birrell and Nelson (1984).

Computer architectures are changing rapidly, and it appears that traditional sequential architectures will be overtaken by newer architectures which use multiple processors in a close network configuration. Hence much work is being directed at how best to utilize such architectures, and at the problems that need solving, including that of making them dependable. These include the problems of making dependable shared buses and shared memory access. Many particular network configurations have been examined in detail including loop, tree, binary cube and hypercube networks. See Pradhan (1986b) and Livingston *et al.* (1987) for further details.

**Useful References:** Bose and Metzner (1986), Tanenbaum (1981), Birman and Joseph (1986), Drummond (1986), Shrivastava and Panzieri (1982), Pradhan (1986b)

## 7.3 Dependable Data

Data plays an important part in any computer system, and a vital part in some applications. Since data is stored in memory or on disc, it is important to ensure that these are as dependable as possible in order to make the data dependable. Coding theory techniques, as mentioned, play an important part in making the memory or disc dependable (see Bose and Metzner, 1986), and yet these techniques may not make the data as dependable as we require. In particular, access to memory may be dependant upon use of an associated

processor which may not be so dependable. Thus it is often important that the data is replicated in different memories or discs. A common abstraction used to hide this replication from the programmer is that of *stable storage*, which allows the programmer to assume that the memory they are using is dependable enough for them not to worry about it. See, for example, Banatre, Muller and Banatre (1987) for details of the construction of a stable storage device, and its integration into a multiprocessor architecture.

A classic example of this data replication is in distributed databases, where a copy of the database is held at different sites or, more generally, copies of parts of the database are spread over a set of sites. This replication helps make the data dependable, but it is important to ensure that accesses to the data are also dependable. A common way of achieving this is by structuring data accesses in terms of *transactions* which comprise a set of reads and/or writes which are atomic in the sense that either the transaction succeeds and the operations are performed correctly, or the transaction fails and none of the operations are effectively performed and so the data does not get corrupted. See Kohler (1981) and Gray *et al.* (1981) for a survey of the problems particular to the area of database systems and an evaluation of a recovery mechanism based on transactions.

Replication need not be at different sites, but within the same memory or disc; for example, a data structure may have added redundant structural data. This allows us to achieve *structural dependency* and *semantic dependency*, that is the dependability of the structure (e.g. that pointers are correct) and of the data. Taylor, Morgan and Black (1980) introduce this idea as applied to list structures, and present some results.

The notion of *data objects* is important in many applications, especially with the rise in popularity of object–oriented programming. An example from distributed operating systems is that of global naming whereby files (the objects) have to be consistent between sites, and the system must be able to tolerate site failures. One solution is that proposed by Mann (1987) who introduces the notion of decentralized naming, and different scopes of names in order to increase performance. *Atomic objects*, typed data objects that provide their own synchronization and recovery, are used in many applications. The extension of C++ classes (for defining new data types) to atomic objects has been suggested by Herlihy and Wing (1988) and by Dixon and Shrivastava (1986). On a wider scale, an object–based view has been integrated into complete distributed systems, such as ISIS (Birman, 1986) and Argus (Liskov, 1988). The architecture being proposed by the ANSA project (sponsored by major UK companies) is also taking a similar approach (Warne, 1988).

**Useful References:** Banatre, Muller and Banatre (1987), Kohler (1981), Birman (1986)

## 7.4 Dependable Processes

Data is manipulated by *processes* and it is important to make these processes dependable. Processes here includes the idea of individual programs, but is a more general term suited to a distributed environment. The use of exceptions and checkpointing for error recovery has already been mentioned briefly, and they can be naturally applied to single processes. Exceptions have been developed more for software error recovery, with facilities for dealing with them being included in many recent languages such as Ada. However, this notion can be extended to deal with process error recovery although it is less applicable here since the errors are less predictable. They are suitable for applications, for example, where it may be alright to proceed to some general input routine after issuing an error message (as in a very simple operating system).

Checkpointing is a commonly used technique in commercial systems. Tandem systems, for example, cause a backup process to be started from the last checkpoint if an error occurs (Serlin, 1984). Efficient application of checkpointing requires a high degree of programming skill and some understanding of system details. This requirement is lessened in some schemes where recovery points are inserted and altered automatically at run–time, as in the scheme proposed by Kim, You and Abouelnaga (1986). A complete system for making processes dependable based on the use of checkpointing and rollback has been proposed by Koo (1987). A programming construct allowing the use of both forwards and backwards recovery techniques has been proposed by Jalote and Campbell (1984).

There are extra problems with these techniques when applied to more than one process. In particular, the state recovered to must be consistent for **all** the processes, which is not so easy when the processes can communicate and so affect the execution of each other. When using checkpoints in such a situation and a process fails and initiates rollback, it will force the processes it has communicated with since the last checkpoint to rollback also. These in turn will cause other processes, perhaps including the original process, to rollback. This may lead to an avalanche of rollbacks until the initial state is the only consistent state

to recover to, which is not typically very useful. Such an event is termed the "domino effect", and many algorithms have been designed to overcome this, such as those of Russell (1980) and Koo and Toueg (1987a).

A useful abstraction in the context of processes is that of fail stop processors (FSPs). A FSP is a processor which when it fails will simply stop, and the failure can be detected. On failure the internal processor state and the contents of volatile (as opposed to stable) storage are lost. FSPs are built using replication in order to be reasonably dependable. True FSPs cannot be fully implemented with finite hardware since it is not possible to detect all faults, but they can be approximated to any desired degree (e.g. see Schlichting, 1982). One method of programming FSPs proposed by Schlichting and Schneider (1981) is by use of Fault–Tolerant Actions (FTAs) which are a program structure involving some *action* and some *recovery* code. On a fault the process is restarted on a new FSP with the state restored to that at the start of the FTA (saved using checkpoints) and the recovery code executed. A program is typically broken into a set of FTAs.

Error compensation techniques are also widely used, typically based on the state machine approach (Schneider, 1986). This is a general method where computation is replicated on processors that are physically and electrically isolated, and permits the effects of errors to be masked by voting on the outputs of these replicas. Triple–Modular Redundancy (TMR), and the generalisation to N-Modular Redundancy (NMR) are common examples of this scheme. Such systems have the advantages over error recovery systems in that there is little time overhead, and the programmer is not forced to think in terms of a particular programming abstraction (such as FTAs).

**Useful References:** Koo (1987), Serlin (1984)

## 7.5 Dependable Software

Making software dependable is to do with being able to tolerate design faults (which includes the common idea of program "bugs"). Typically this is performed by verification and validation (V&V) procedures, including testing procedures, and it is important that these procedures are performed throughout the software life cycle. See Goel and Bastani (1985) and the associated papers for an introduction to some of the work being done.

One widely used technique for tolerating a limited set of faults in a fairly simple way is that of *exceptions*. For example, Ada provides handling of divide by zero and overflow faults among others. Once a fault has occurred program execution is automatically transferred to an exception handler which provides application dependant recovery. As mentioned, this idea can be extended to the notion of exceptional domains for error detection and for the tolerance of physical faults.

Another idea that has received attention is that of *design diversity*, the basic idea of which is to design different versions of software that meet a given specification, and then select in some way from the results of executing each version to find a dependable result. The versions are typically written by different people, or in a different language, or using different algorithms. Techniques for design diversity can be in the form of error recovery *or* error compensation. An example of an error recovery technique is that of *recovery blocks* developed at Newcastle in the 1970's (Randell, 1975). Using this technique an initial version of the software is executed, and then an *acceptance test* is made to see if the results are in some sense acceptable (i.e. that no faults have occurred). If not then a second version of the software is executed in the same starting state as for the first version (saved using checkpointing), and the acceptance test made again. This is repeated until the result is acceptable or the system fails with no more possible versions to try. A key problem is in the choice of acceptance test which needs to be made carefully; this is discussed by Hecht and Hecht (1986). The "N-Version Programming" project at UCLA is examining an error compensation technique for design diversity. Here the different versions are executed concurrently with a majority decision on the output – this is a very natural extension of the state machine approach discussed earlier, and the two can be usefully combined.

**Useful References:** Goel and Bastani (1985), Anderson (1987), Avižienis (1985), Randell (1975)

## 7.6 Agreement Algorithms

Algorithms for achieving agreement between a set of processes are important for many fault–tolerance methods. The problem of agreement occurs as the "transaction commit problem" in distributed data bases, where a set of data manager processes need to decide whether to accept or discard the results of a transaction.

It also occurs when using the state machine approach to fault–tolerance, as described earlier, for agreeing values between a set of processes. Unfortunately simple majority voting schemes are not sufficient for dealing with all possible kinds of fault. Discussions of the problems and a proposed solution was given in the pioneering paper of Pease, Shostak and Lamport (1980) which arose out of the SIFT project (Wensley *et al.*, 1978) that used a state machine approach for designing a fault tolerant flight control system. The problem of agreement comes in many flavours, such as *interactive consistency* used by the previous authors, and more generally as the Byzantine Generals Problem: given a group of generals of the Byzantine army camped with their troops around an enemy city, the generals must agree on a common battle plan communicating only by messanger. However, some of the generals may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement (Lamport, Shostak and Pease, 1982). Various solvability results exist for the problem and its variants. In particular, using only oral messages in a synchronous system, the problem is solvable if and only if more than two–thirds of the generals are loyal. But with *authenticated* messages (which can't be forged by traitors) the problem is solvable for any number of generals and possible traitors. However, within an asynchronous system it is not possible to design a terminating protocol to solve this problem, even if there is only one traitor (Fischer, Lynch and Paterson, 1985). In this case it is possible to design protocols to solve the problem which may never terminate, but this would occur with probability zero, and the expected termination time is finite. Two such classes of protocols are *randomized* protocols (such as given by Perry, 1985) and *probabilistic* protocols (such as given by Bracha and Toueg, 1983).

The form of agreement termed "inexact agreement" is useful in performing *clock synchronization*, that is keeping the clocks of a set of processes arbitrarily close to each other. This is a weaker form of agreement than Byzantine Generals, and for a given level of reliability is cheaper to achieve. The performance of clock synchronization protocols is measured in terms of *precision* – how close together the clocks are, and *accuracy* – how close the clocks are to "real-time". It may also be useful that such protocols have graceful degradation when there are too many faults for them to achieve the desired performance (see Mahaney and Schneider, 1985). It has been shown by Schneider, 1987 that current dependable clock synchronization protocols result from refining a single paradigm that is based on having a dependable time source that periodically issues messages to cause processes to synchronize their clocks. An important choice in refining this paradigm is of a convergence function, which brings the clocks closer together based on their current values (taking the average being a simple such function). In a system which uses broadcasts, clock synchronization can be performed at little extra cost (Babaoğlu and Drummond, 1986).

**Useful References:** Lamport, Shostak and Pease (1982), Pease, Shostak and Lamport (1980), Dolev (1982), Fischer, Lynch and Paterson (1985), Perry (1985), Schneider (1987)

# 8 Conclusions

There is an increasing need for computer systems to be dependable as they are being increasingly used in safety–critical applications, and in applications where a system breakdown is highly undesirable. Special purpose systems have been designed for specific applications, such as for the Space Shuttle and the Voyager spacecraft (Siewiorek and Swarz, 1982), and it will probably remain the case for some time that such "special" applications will be purpose–designed. However, for many applications a more general purpose dependable architecture is appropriate. Most commercial machines concentrate on providing state machine and checkpoint based recovery mechanisms (Serlin, 1984), with newer research ideas only slowly filtering through. Currently fault tolerant machines are manufactured by specialist companies such as Tandem, but there is increasing interest by the established companies (e.g. IBM) and so this will probably change.

Dependability methods are maturing from a collection of *ad hoc* techniques into more complete approaches to the problem within the larger context of distributed systems. Although many of the basic problems have been solved there is a need to see which are efficient solutions, and which solutions are suited to particular applications. There is also a need for a formal framework within which fault tolerance can be reasoned about. The work in developing more general distributed systems (including dependability aspects) will ultimately affect the way we view computer systems and the way we program. Yet the design of dependable systems is still a very open area of research.

# A Annotated References

This appendix presents a set of annotated references for many of the areas discussed. They are grouped into sections according to the presentation given in Section 7, and include some of the references cited there. In addition there is a section giving details of some other surveys of the area which take different approaches than in this report, and also a section giving some of the work in formal methods that is related to dependability.

## A.1 Surveys

Anderson and Lee (1981).
Describes a general approach to the principles and practice of fault–tolerance, based on a framework of error detection, damage confinement and assessment, error recovery and continued service. This framework is used to view existing approaches to achieving dependability. It includes a useful set of references and an annotated bibliography.

Avižienis (1976).
A good introduction to the basic concepts and techniques used in achieving dependability. Topics covered include fault classification, redundancy techniques, reliability modeling and some examples of dependable systems.

Borning (1987).
A very interesting look at the problems of dependability in command and control systems for nuclear weapons. Details of system requirements, actual failures, and approaches to the problems are presented.

Paul and Siegert, editors (1985).
A tutorial style introduction to distributed systems, including dependability aspects, from a semi–formal point of view. Includes consideration of specifications, language constructs and paradigms.

Pradhan, editor (1986a).
A selection of papers introducing fault–tolerance methods. The main emphasis is on the problems encountered at the circuit level, and the solutions to these problems.

Randell, Lee and Treleaven (1978).
A definitive survey of the problems involved in achieving dependable systems. Topics covered include hardware and software redundancy, atomic actions, error detection, damage assessment, and forward and backward recovery techniques. Three specific systems are examined in detail.

Serlin (1984).
A very good introduction to the dependable commercial architectures which have been developed. Includes a detailed summary of 11 currently available systems. Concludes with some open issues for the future.

Siewiorek and Swarz, editors (1982).
A collection of papers covering the theory and practice of dependable system design. All the essential concepts are presented in depth, from a practical viewpoint. Twelve computer systems are outlined, including some for aircraft and spacecraft control. A good source of background material.

Stankovic (1984).
A survey of six fundamental issues in distributed computer systems: the object model, access control, distributed control, dependability, heterogeneity and efficiency. It places dependability issues into a wider perspective, and is a good source of general references.

## A.2 Dependable Networks

Babaoğlu, Drummond and Stephenson (1986).
The execution time of dependable broadcast protocols is characterized as a function of the properties of the underlying communications network. A protocol is derived that implements dependable broadcasts for any member of a class of network structures. A novel proof technique based on graph mappings is used to obtain some lower bound results.

Birman and Joseph (1986).
The design of communication primitives is discussed, and a collection of such primitives are described which facilitate the implementation of the fault–tolerant process groups used in the ISIS system.

Birrell and Nelson (1984).
The design of remote procedure calls is discussed, and one particular RPC package and its use is described in some depth. Some optimizations for high performance and low load are presented.

Chang and Maxemchuck (1984).
A reliable broadcast protocol for an unreliable network (of fail–stop processors and lossy communications) is described. It guarantees that all operational receivers receive the same messages and in the same order.

Chlamtac and Kutten (1985).
A good introduction to broadcasts in radio networks including a brief survey of existing methods, and a graph–oriented model within which to view the problems involved.

Drummond (1986).
Time complexity of dependable broadcasts is characterized with respect to communication models. Dependable broadcast protocols are developed in systems where communication is through multicasts.

Shrikanth (1986).
This thesis presents a way of developing algorithms under a simple model of failure (assuming authentification of messages) in which algorithms are simple and easy to derive, and then translating them into a form using broadcast primitives which achieve authentication without using digital signatures.

Shrivastava and Panzieri (1982).
The semantics of an RPC mechanism is given, together with an overview of issues involved in designing RPCs. Algorithms for one particular RPC mechanism are given.

Tanenbaum (1981).
The definitive work presenting the different issues involved in computer networks. Networks are discussed with respect to the ISO reference model, bringing together all the basic concepts.

## A.3 Dependable Data

Birman (1986).
A good overview of the ISIS system, introducing and motivating all the essential concepts. The performance of a prototype is given for a variety of system loads and configurations.

Birman and Joseph (1987).
A new software abstraction called the virtually synchronous process group is introduced and applied to problems such as group remote procedure calls, global task decomposition and dynamic reconfiguration. A new version of ISIS based on this abstraction is described.

Birman, Joseph and Stephenson (1986).
The use of bulletin boards in loosely coupled distributed systems is considered. The notion of a consistent behavior when unreliable processes concurrently access a bulletin board is formalized. Techniques for implementing bulletin boards are presented.

Kohler (1981).
Solution techniques to two problems are surveyed. The first is synchronizing access to shared objects, a generalization of the concurrency control problem in database systems. The second is the recovery of objects in spite of errors (due to the use, the application or the system). A solution using atomic actions is put forward.

Liskov (1988).
The Argus system is presented by means of the example of a dependable distributed banking application.

Nett, Kröger and Kaiser (1986).
The recovery aspects of a distributed operating system based on a layered, generalized transaction concept in an object–oriented environment are discussed. The recovery mechanisms are based on the use of graphs linking dependencies between concurrent actions.

Schwarz and Spector (1984).
Transactions are introduced and the synchronization issues that arise when transaction facilities are extended for use with shared abstract data types are discussed. The use of non–serializable transaction execution is examined within a formal framework.

Shrivastava, Mancini and Randell (1987).
It is shown how an object and action based model is the dual of a process and message based model (which uses conversations). Thus there appears no inherent reason for favoring one approach over the other.

Verhofstad (1978).
A survey of techniques and tools for recovery in filing systems, database systems and operating systems. A categorization of recovery techniques in database systems is given.

## A.4 Dependable Processes

Jalote and Campbell (1984).
CSP is extended with an S-Conversion construct which supports both forward and backward error recovery. It is shown how this can be implemented using the existing CSP primitives.

Joseph, Moitra and Soundararajan (1984).
A proof system is constructed for a CSP based language extended with fault alternatives for the specification of alternative code to be executed on errors detected at communication time. The method is applied to a bounded buffer example.

Kim, You and Abouelnaga (1986).
A scheme is presented for efficient backward recovery in loosely coupled networks. It involves programmer–transparent coordination of recovery points, with recovery points automatically being inserted and deleted at run–time, and avoiding the domino effect.

Koo (1987).
A checkpointing/rollback mechanism is introduced that allows distributed programs to be written assuming processors do not fail, and can then execute correctly on systems where transient processor failures can occur. The mechanism is transparent to the programmer.

Mancini and Pappalardo (1988).
The CSP notation is used to describe an NMR system, formalizing the notion of replication and showing an NMR configuration correct.

Russell (1980).
An important paper on checkpointing. Recovery primitives are defined to perform checkpointing and rollback in a class of asynchronous systems. Different types of propagation of state restoration are identified. Sufficient conditions are developed for a system to be domino–free.

Schlichting (1982).
A method is presented, based on the use of fail–stop processors, for designing programs that can cope with system errors. Axiomatic program verification techniques are extended and applied to the two–phase commit protocol as an example.

Schlichting and Schneider (1983).
The standard introduction to fail–stop processors, giving an implementation, a method of programming using fault–tolerant actions. It also discusses termination, response time and process–control software.

Schneider (1986).
A readable introduction to the state machine approach, presented with regard to one particular style of implementation. All the relevant concepts and implementation details are discussed.

Schneider and Lamport (1985).
Some paradigms associated with programming distributed systems are described, including various agreement protocols, the state machine approach (in a different form to the above) and computation of global states. The paper is tutorial in nature.

Shrivastava (1986).
A tutorial paper briefly introducing redundancy and Byzantine Agreement, and then concentrating on the NMR approach, giving the concepts and problems involved.

Thanawastien, Pamula and Varol (1986).
The sequential and lookahead rollback schemes are investigated and compared within a fault model. The optimal number of checkpoints to minimize the cost of rollback in each scheme is given.

## A.5 Dependable Software

Anderson (1987).
Provides a brief, but good, survey of the work that has been done, including practical examples, in the area of design diversity.

Avižienis and Kelly (1984).
Design diversity is introduced. Experiments to test the usefulness of this concept are detailed, together with future research directions.

Avižienis (1985).
A more detailed version of the above, describing N-Version Programming in detail. Requirements for implementing N-Version software are given, and a testbed for the execution of such software described.

Goel and Bastani (1985).
This forms a preface to two volumes of the IEEE Transactions on Software Engineering which concentrate on the problems involved in making dependable software.

Randell (1975).
A method for structuring systems by use of recovery blocks, conversations and fault tolerant interfaces is presented and motivated. The aim is to provide dependable error detection and recovery facilities which can cope with design faults, although applicable to physical faults. The paper presents an introduction to the structuring of complex systems.

## A.6 Agreement Algorithms

Bar-Noy and Peleg (1986).
It is shown that some weak forms of processor cooperation are achievable in a completely asynchronous system, despite the impossibility result of Fischer, Lynch and Paterson (1985). Processor renaming is used as an example.

Cristian, Aghili and Strong (1986).
A clock synchronization protocol is presented which tolerates any number of omission and performance faults. Arbitrary numbers of simultaneous processor joins are handled.

Fischer, Lynch and Paterson (1985).
Presents an important result – that in an asynchronous system there is no finite agreement protocol that can tolerate even one fail–stop process.

Koo and Toueg (1987b).
It is shown that for asynchronous (and hence synchronous) systems, protocols that guarantee knowledge gain by message passing cannot be guaranteed to terminate, even if it is assumed that only transient communication failures can occur and that a proof of weak termination is sufficient.

Lamport, Shostak and Pease (1982).
A comprehensive introduction to the Byzantine Generals problem with its variants; includes useful results and algorithms.

Mahaney and Schneider (1985).
Two fault tolerant protocols for inexact agreement are presented and their application to clock synchronization is explained. Accuracy, precision and graceful degradation results are derived for the protocols.

Pease, Shostak and Lamport (1980).
A pioneering paper which defines the notion of interactive consistency and presents protocols to achieve it within a faulty environment. Some results are also given.

Perry (1985).
Early stopping protocols for the Byzantine Generals problem are presented under a variety of system models. Multi–valued agreement is considered. A scheme to solve the Byzantine Generals problem that is independent of the fault model is given.

## A.7 Formal Methods

Chandy and Misra (1988).
Presents a new theory of programming which transcends differences in architectures by concentrating on specifications

and their refinement. It is applied to specifying dependable channels and communication protocols, and the Byzantine Generals problem.

Cohn (1987).
Illustrates the use of higher–order logic in proving some correctness properties of the Viper processor.

Cristian (1985).
Axiomatic reasoning methods are extended to the design of dependable systems by modeling faults as operations that are performed at random time intervals. Design correctness is proven using a programming logic extended with fault axioms and rules. Stochastic modeling is employed to verify reliability and availability properties.

Fischer and Zuck (1988).
A temporal logic is presented for reasoning about uncertainty in distributed systems containing both probabilistic and nondeterministic transitions. The logic is applied to the coordinated attack problem and to the authenticated agreement problem.

Herlihy and Wing (1987).
A means of specifying graceful degradation is described. A lattice of specifications is used with a set of constraints where the stronger these constraints, the more restrictive the specification. Examples are given for programs tolerating site crashes, timing anomalies and synchronization conflicts.

Jifeng and Hoare (1987).
An algebraic specification (in CSP) is given for a simple algorithm to allow a process to overcome an internal failure by use of checkpoints. The algorithm is shown correct by the application of algebraic laws.

Melliar–Smith and Schwartz (1982).
The formal specification and proof method used to show that the SIFT computer meets its requirements is described with respect to hardware and software. A hierarchy of design specifications is given with mappings between the levels. This provides a useful illustration of the use of formal methods in a practical situation.

## Cited References


T. Anderson (1987).
Design fault tolerance in practical systems. In B. Littlewood, editor, *Software Reliability*, chapter 4, pages 44–55, Blackwell Scientific Publications.

T. Anderson and P.A. Lee (1981).
*Fault Tolerance Principle and Practice*. Prentice Hall.

T. Anderson and P.A. Lee (1982).
Fault–tolerance terminology proposals. In *Digest of FTCS–12*, pages 29–33.

A. Avižienis (1976).
Fault-tolerant systems. *IEEE Transactions on Computers*, C-25(12).

A. Avižienis (1985).
The *N*–version approach to fault–tolerant software. *IEEE Transactions on Software Engineering*, SE–11(12):1491–1501.

A. Avižienis and J. Kelly (1984).
Fault tolerance by design diversity: concepts and experiments. *Computer*, 17(8):67–80.

O. Babaoğlu and R. Drummond (1986).
*(Almost) No Cost Clock Synchronization*. Technical Report 86-791, Department of Computer Science, Cornell University.

Ö. Babaoğlu, R. Drummond, and P. Stephenson (1986).
*Reliable Broadcast Protocols and Network Architecture: Tradeoffs and Lower Bounds*. Technical Report 86-754, Department of Computer Science, Cornell University.

J.M. Bacon and K.G. Hamilton (1987).
*Distributed Computing with RPC: The Cambridge Approach*. Technical Report 117, Computer Laboratory, University of Cambridge.

M. Banatre, G. Muller, and J-P. Banatre (1987).
*An Active Stable Storage and its Integration in a Multiprocessor Architecture*. Internal Report 362, IRISA. (also INRIA Report 693).

A. Bar-Noy and D. Peleg (1986).
*Processor Renaming in Asynchronous Environments*. Report No. STAN-CS-86-1131, Department of Computer Science, Stanford University.

E. Best and F. Cristian (1981).
Systematic detection of exception occurrences. *Science of Computer Programming*, 1:115–144.

K.P. Birman (1986).
*ISIS: A System for Fault-Tolerant Distributed Computing*. Technical Report 86-744, Department of Computer Science, Cornell University.

K.P. Birman and T.A. Joseph (1986).
*Communication Support for Reliable Distributed Computing*. Technical Report 86-753, Department of Computer Science, Cornell University.

K.P. Birman and T.A. Joseph (1987).
*Exploiting Virtual Synchrony in Distributed Systems*. Technical Report 87-811, Department of Computer Science, Cornell University.

K.P. Birman, T.A. Joseph, and P. Stephenson (1986).
*Programming with Shared Bulletin Boards in Asynchronous Distributed Systems*. Technical Report 86-772, Department of Computer Science, Cornell University.

A.D. Birrell and B.J. Nelson (1984).
Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59.

A. Borning (1987).
Computer system reliability and nuclear war. *Communications of the ACM*, 30(2):112–131.

B. Bose and J. Metzner (1986).
Coding theory for fault–tolerant systems. In D.K. Pradhan, editor, *Fault–Tolerant Computing: Theory and Techniques*, chapter 4, Prentice-Hall.

G. Bracha and S. Toueg (1983).
*Asynchronous Consensus and Byzantine Protocols in Faulty Environments.* Technical Report 83-559, Department of Computer Science, Cornell University.

K.M. Chandy and J. Misra (1988).
*Parallel Program Design: A Foundation.* Addison Wesley.

J-M. Chang and N.F. Maxemchuk (1984).
Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273.

I. Chlamtac and S. Kutten (1985).
On broadcasting in radio networks - problem analysis and protocol design. *IEEE Transactions on Communications*, COM-33(12):1240–1246.

A. Cohn (1987).
*A Proof of Correctness of the Viper Microprocessor: The First Level.* Technical Report 104, Computer Laboratory, University of Cambridge.

F. Cristian (1985).
A rigorous approach to fault–tolerant programming. *IEEE Transactions on Software Engineering*, SE–11(1):23–31.

F. Cristian, H. Aghili, and R. Strong (1986).
Clock synchronization in the presence of omission and performance faults, and processor joins. In *Digest of FTCS–16*, pages 218–223, IEEE.

E.W. Czeck, D.P. Siewiorek, and Z. Segall (1985).
*Fault Free Performance Validation of a Fault-Tolerant Multiprocessor: Baseline and Synthetic Workload Measurements.* Technical Report CMU-CS-85-177, Department of Computer Science, Carnegie–Mellon University.

G.N. Dixon and S.K. Shrivastava (1986).
*Exploiting Type Inheritance Facilities to Implement Recoverability in Object Based Systems.* Technical Report 223, Computing Laboratory, University of Newcastle upon Tyne.

D. Dolev (1982).
The byzantine generals strike again. *Journal of Algorithms*, 3:14–30.

D. Dolev and H.R. Strong (1983).
Authenticated algorithms for byzantine agreement. *SIAM Journal of Computing*, 12(4):656–666.

R. Drummond (1986).
*Impact of Communication Networks on Fault-Tolerant Distributed Computing.* PhD thesis, Department of Computer Science, Cornell University. TR 86-748.

A.M. Farley (1987).
*Reliable Minimum–Time Broadcast Networks.* Technical Report CIS-TR-87-06, Department of Computer and Information Science, University of Oregon.

M.J. Fischer, N.A. Lynch, and M.S. Paterson (1985).
Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

M.J. Fischer and L.D. Zuck (1988).
Reasoning about uncertainty in fault–tolerant distributed systems. In *Formal Techniques in Real–Time and Fault–Tolerant Systems*, Springer Verlag. LNCS 331.

A.L. Goel (1985).
Software reliability models: assumptions, limitations, and applicability. *IEEE Transactions on Software Engineering*, SE-11(12):1411–1423.

A.L. Goel and F.B. Bastani (1985).
Software reliability. *IEEE Transactions on Software Engineering*, SE-11(12):1409–1410.

J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger (1981).
The recovery manager of the system R database manager. *ACM Computing Surveys*, 13(2):223–242.

H. Hecht and M Hecht (1986).
Fault–tolerant software. In D.K. Pradhan, editor, *Fault–Tolerant Computing: Theory and Techniques*, chapter 10, Prentice-Hall.

M.P. Herlihy and J.M. Wing (1988).
Reasoning about atomic objects. In M. Joseph, editor, *Formal Techniques in Real–Time and Fault–Tolerant Systems*, Springer Verlag. LNCS 331.

M.P. Herlihy and J.M. Wing (1987).
*Specifying Graceful Degradation in Distributed Systems.* Technical Report CMU-CS-87-120, Computer Science Department, Carnegie Mellon University.

P. Jalote and R.H. Campbell (1984).
Fault tolerance using communicating sequential processes. In *Digest of FTCS–14*, IEEE.

H. Jifeng and C.A.R. Hoare (1987).
Algebraic specification and proof of a distributed recovery algorithm. *Distributed Computing*, 2:1–12.

M. Joseph, A. Moitra, and N. Soundararajan (1984).
*Proof Rules for Fault Tolerant Distributed Programs.* Technical Report 84-643, Department of Computer Science, Cornell University.

K.H. Kim, J.H. You, and A. Abouelnaga (1986).
A scheme for coordinated execution of independently designed recoverable distributed processes. In *Digest of FTCS–16*, pages 130–135, IEEE.

W.H. Kohler (1981).
A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Computing Surveys*, 13(2):149–184.

R. Koo (1987).
*Techniques for Simplifying the Programming of Distributed Systems.* PhD thesis, Department of Computer Science, Cornell University. TR 87-858.

R. Koo and S. Toueg (1987a).
Checkpointing and rollback–recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31.

R. Koo and S. Toueg (1987b).
*Effects of Message Loss on Distributed Termination.* Technical Report 87-823, Department of Computer Science, Cornell University.

C.J. Koomen (1985).
Algebraic specification and verification of communication protocols. *Science of Computer Programming*, 5:1–36.

L. Lamport, R. Shostak, and M. Pease (1982).
The byzantine generals problem. *ACM TOPLAS*, 4(3):382–401.

J.C. Laprie (1985).
Dependable computing and fault tolerance: concepts and terminology. In *Digest of FTCS–15*, pages 2–11.

N.G. Leveson (1986).
Software safety: what, why, and how. *ACM Computing Surveys*, 18(2):125–164.

B. Liskov (1988).
Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312.

M. Livingston, Q. Stout, N. Graham, and F. Harary (1987).
*Subcube Fault–Tolerance in Hypercubes.* Technical Report CRL-TR-12-87, Computing Research Laboratory, University of Michigan.

S.R. Mahaney and F.B. Schneider (1985).
*Inexact Agreement: Accuracy, Precision, and Graceful Degradation.* Technical Report 85-683, Department of Computer Science, Cornell University.

L. Mancini and G. Pappalardo (1988).
Towards a theory of replicated processing. In M. Joseph, editor, *Formal Techniques in Real–Time and Fault–Tolerant Systems*, Springer Verlag. LNCS 331.

T.P. Mann (1987).
*Decentralized Naming in Distributed Computer Systems.* PhD thesis, Department of Computer Science, Stanford University.
STAN-CS-87-1179.

E.J. McCluskey (1986).
Design for testability. In D.K. Pradhan, editor, *Fault–Tolerant Computing: Theory and Techniques*, chapter 2, Prentice-Hall.

S. McConnel and D.P. Siewiorek (1982).
Evaluation criteria. In D.P. Siewiorek and R.S. Swarz, editors, *Theory and Practice of Reliable System Design*, chapter 5, Digital Press.

P.M. Melliar-Smith and R.L. Schwartz (1982).
Formal specification and mechanical verification of SIFT: a fault-tolerant flight control system. *IEEE Transactions on Computers*, C-31(7):616–630.

E. Nett, R. Kröger, and J. Kaiser (1986).
Implementing a general error recovery mechanism in a distributed operating system. In *Digest of FTCS–16*, pages 124–129, IEEE.

M. Paul and H.J. Siegert, editors (1985).
*Distributed Systems.* Springer Verlag, LNCS 190.

M. Pease, R. Shostak, and L. Lamport (1980).
Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234.

K.J. Perry (1985).
*Early Stopping Protocols For Fault-Tolerant Distributed Agreement.* PhD thesis, Department of Computer Science, Cornell University. TR 85-662.

D. Pountain (1988).
Fast track vs. failsafe. *Byte*, July.

D.K. Pradhan, editor (1986a).
*Fault–Tolerant Computing: Theory and Techniques, Volumes 1 & 2.* Prentice-Hall.

D.K. Pradhan (1986b).
Fault–tolerant multiprocessor and vlsi–based system communication architectures. In D.K. Pradhan, editor, *Fault–Tolerant Computing: Theory and Techniques*, chapter 7, Prentice-Hall.

B. Randell (1975).
System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE–1(2):220–232.

B. Randell, P.A. Lee, and P.C. Treleaven (1978).
Reliability issues in computing system design. *ACM Computing Surveys*, 10(2):123–165.

D.L. Russell (1980).
State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194.

R.D. Schlichting (1982).
*Axiomatic Verification to Enhance Software Reliability.* PhD thesis, Department of Computer Science, Cornell University. TR 82-480.

R.D. Schlichting and F.B. Schneider (1981).
*An Approach to Designing Fault-Tolerant Computing Systems.* Technical Report 81-479, Department of Computer Science, Cornell University.

R.D. Schlichting and F.B. Schneider (1983).
Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computing Systems*, 1(3):222–238.

F.B. Schneider (1986).
*The State Machine Approach: A Tutorial.* Technical Report 86-800, Department of Computer Science, Cornell University. (revised June 1987).

F.B. Schneider (1982).
Synchronization in distributed systems. *ACM TOPLAS*, 4(2):125–148.

F.B. Schneider (1987).
*Understanding Protocols for Byzantine Clock Synchronization.* Technical Report 87-859, Department of Computer Science, Cornell University.

F.B. Schneider and L. Lamport (1985).
Paradigms for distributed programs. In *Distributed Systems*, chapter 8, Springer Verlag. LNCS 190.

P.M. Schwarz and A.Z. Spector (1984).
Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250.

O. Serlin (1984).
Fault–tolerant systems in commercial applications. *Computer*, 17(8):19–30.

S.K. Shrivastava (1986).
*Replicated Distributed Processing.* Technical Report 222, Computing Laboratory, University of Newcastle upon Tyne.

S.K. Shrivastava, L. Mancini, and B. Randell (1987).
*On the Duality of Fault Tolerant System Structures.* Memorandum SRM/455, Computing Laboratory, University of Newcastle upon Tyne.

S.K. Shrivastava and F. Panzieri (1982).
The design of a reliable remote procedure call mechanism. *IEEE Transactions on Computers*, C-31(7):692–697.

D.P. Siewiorek and R.S. Swarz, editors (1982).
*The Theory and Practice of Reliable System Design.* Digital Press.

T.K. Srikanth (1986).
*Designing Fault-Tolerant Algorithms for Distributed Systems Using Communication Primitives.* PhD thesis, Department of Computer Science, Cornell University. TR 86-739.

J.A. Stankovic (1984).
A perspective on distributed computer systems. *IEEE Transactions on Computers*, C-33(12):1102–1115.

J.J. Stiffler (1986).
Computer–aided reliability estimation. In D.K. Pradhan, editor, *Fault–Tolerant Computing: Theory and Techniques*, chapter 9, Prentice-Hall.

A.S. Tanenbaum (1981).
*Computer Networks*. Prentice Hall.

D.J. Taylor, D.E. Morgan, and J.P. Black (1980).
Redundancy in data structures: improving software fault tolerance. *IEEE Transactions on Software Engineering*, SE-6(6):585–594.

S. Thanawastien, R.S. Pamula, and Y.L. Varol (1986).
Evaluation of global checkpoint rollback strategies for error recovery in concurrent processing systems. In *Digest of FTCS–16*, pages 246–251, IEEE.

M. Thomas (1988).
*Should we Trust Computers?* BCS/UNISYS Annual Lecture, Praxis Systems plc, Bath, U.K..

Y. Tohma (1986).
Coding techniques in fault–tolerant, self–checking, and fail–safe circuits. In D.K. Pradhan, editor, *Fault–Tolerant Computing: Theory and Techniques*, chapter 5, Prentice-Hall.

J. Verhofstad (1978).
Recovery techniques for database systems. *ACM Computing Surveys*, 10(2):167–.

J. Warne (1988).
*Getting Fault–Tolerance Theory into Practice (Workshop Notes)*. Technical Report PR.67.00, ANSA Project.

J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock (1978).
SIFT: design and analysis of a fault–tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255.

## Other References


T. Anderson, P.A. Barrett, D.N. Halliwell, and M.R. Moulding. An evaluation of software fault tolerance in a practical system. In *Digest of FTCS–15*, pages 140–145, 1985.

T. Anderson and J.C. Knight. A framework for software fault tolerance in real–time systems. *IEEE Transactions on Software Engineering*, SE-9(3):355–364, 1983.

T. Anderson and J.C. Knight. *Practical Software Fault Tolerance for Real-Time Systems.* Technical Report 169, University of Newcastle upon Tyne, 1981.

Ö. Babaoğlu. *Engineering Fault-Tolerant Distributed Computing Systems.* Technical Report 86-755, Department of Computer Science, Cornell University, 1986.

Ö. Babaoğlu. *On the Reliability of Fault-Tolerant Distributed Computing Systems.* Technical Report 86-738, Department of Computer Science, Cornell University, 1986.

R. Bar-Yehuda, S. Kutten, Y. Wolfstahl, and S. Zaks. *Making Distributed Spanning Tree Algorithms Fault-Resilient.* Technical Report, Department of Computer Science, Technion, Israel, 1986.

M. Beck, T.K. Srikanth, and S. Toueg. *Implementation Issues in Clock Synchronization.* Technical Report 86-749, Department of Computer Science, Cornell University, 1986.

T.E. Bihari and K. Schwan. *Dynamic Adaptation of Real–Time Software for Reliable Performance.* Technical Report OSU-CISRC-5/88-TR17, The Ohio State University Computer and Information Science Research Center, 1988.

K.P. Birman and T.A. Joseph. *Exploiting Replication.* Technical Report 88-917, Department of Computer Science, Cornell University, 1988.

K.P. Birman and T.A. Joseph. *Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems.* Technical Report 84-644, Department of Computer Science, Cornell University, 1984.

P.I.P. Boulton and M.A.R. Kittler. Estimating program reliability. *BCS Computer Journal*, 22(4):328–331, 1979.

G. Bracha. *An Asynchronous $\lfloor (n-1)/3 \rfloor$-Resilient Consensus Protocol.* Technical Report 84-590, Department of Computer Science, Cornell University, 1984.

W.J. Cullyer. High integrity computing. In M. Joseph, editor, *Formal Techniques in Real–Time and Fault–Tolerant Systems*, Springer Verlag, 1988. LNCS 331.

J.E. Dobson and B. Randell. *Building Reliable Secure Computing Systems out of Unreliable Insecure Components.* Technical Report 214, Computing Laboratory, University of Newcastle upon Tyne, 1986.

I. Durham and M. Shaw. *Specifying Reliability as a Software Attribute.* Technical Report CMU-CS-82-148, Department of Computer Science, Carnegie–Mellon University, 1982.

C.F. Everhart. *Making Robust Programs.* PhD thesis, Department of Computer Science, Carnegie–Mellon University, 1985. CMU-CS-85-170.

P.D. Ezhilchelvan, S. Shrivastava, and A. Tully. *Constructing Replicated Systems using Processors with Point to Point Communication Links.* Draft Report, Computing Laboratory, University of Newcastle upon Tyne, 1988.

P.D. Ezhilchelvan and S.K. Shrivastava. *A Characterisation of Faults in Systems.* Technical Report 206, Computing Laboratory, University of Newcastle upon Tyne, 1985.

F. Feather, D. Siewiorek, and Z. Segall. *Validation of a Fault-Tolerant Multiprocessor: Baseline Experiments and Workload Implementation.* Technical Report CMU-CS-85-145, Department of Computer Science, Carnegie–Mellon University, 1985.

R. Gupta, A. Zorat, and I.V. Ramakrishnan. A fault–tolerant multipipeline architecture. In *Digest of FTCS–16*, pages 350–355, IEEE, 1986.

A.L. Hopkins, T.B. Smith, and J.H. Lala. Ftmp – a highly reliable fault–tolerant multiprocessor for aircraft. *Proceedings of the IEEE*, 66(10):1221–1239, 1978.

M. Joseph. *Formal Techniques in Real–Time and Fault–Tolerant Systems*. Springer Verlag, 1988. LNCS 331.

J.C. Knight, N.G. Leveson, and L.D. St.Jean. A large scale experiment in n-version programming. In *Digest of FTCS–15*, pages 135–140, 1985.

J.C. Knight and J.I.A. Urquhart. On the implementation and use of Ada on fault–tolerant distributed systems. *IEEE Transactions on Software Engineering*, SE-13(5):553–563, 1987.

H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, C-36(8):933–940, 1987.

J.H. Lala. A byzantine resilient fault tolerant computer for nuclear power plant applications. In *Digest of FTCS–16*, pages 338–343, IEEE, 1986.

L. Lamport. Using time instead of timeout for fault–tolerant distributed systems. *ACM TOPLAS*, 6(2):254–280, 1984.

Y.-H. Lee. *Characterization of Failure Handling in Fault-Tolerant Multiprocessor Systems*. PhD thesis, Computing Research Laboratory, University of Michigan, 1984. CRL-TR-48-84.

B. Liskov. The argus language and system. In M. Paul and H.J. Siegert, editors, *Distributed Systems*, chapter 8, Springer Verlag LNCS 190, 1985.

B. Littlewood. Theories of software reliability: how good are they and how can they be improved? *IEEE Transactions on Software Engineering*, SE-6(5):389–500, 1980.

K. Lodaya. *Proof Theory for Exception Handling in Distributed Programs*. PhD thesis, Tata Institute of Fundamental Research, Bombay, 1987. TR CS-87/30.

J.A. McDermid. *Assurance in High Integrity Software*. YCS 96, Department of Computer Science, University of York, 1987.

J.F. Meyer. *Unified Performance-Reliability Evaluation*. Technical Report CRL-TR-27-84, Computing Research Laboratory, University of Michigan, 1984.

I.M. Nixon. *The Automatic Synthesis of Fault Tolerant and Fault Secure VLSI Systems*. PhD thesis, Department of Computer Science, University of Edinburgh, 1988. CST-50-88.

M.T. Norris, M.W. Shields, and J. Ganeri. A theoretical basis for the construction of interactive systems. *British Telecom Technology Journal*, 5(2), 1987.

S. Osaki and T. Nishio. *Reliability Evaluation of Some Fault–Tolerant Computer Architectures*. Springer Verlag LNCS 97, 1980.

G. Pappalardo and S.K. Shrivastava. A formal treatment of interference in remote procedure calls. In M. Joseph, editor, *Formal Techniques in Real–Time and Fault–Tolerant Systems*, Springer Verlag, 1988. LNCS 331.

S. Pfleger. *Design Concepts for Robust Software*. Technical Report 224, Computing Laboratory, University of Newcastle upon Tyne, 1986.

J. Qian. *An Assertional Proof of a Byzantine Agreement Protocol*. Technical Report 83-566, Department of Computer Science, Cornell University, 1983.

F.B. Schneider. *Abstraction for Fault Tolerance in Distributed Systems*. Technical Report 86-745, Department of Computer Science, Cornell University, 1986.

F.B. Schneider. *Comparison of the Fail-Stop Processor and State Machine Approaches to Fault-Tolerance.* Technical Report 82-533, Department of Computer Science, Cornell University, 1982.

F.B. Schneider and R.D. Schlichting. Towards fault tolerant process control software. *IEEE CH1600-6/81*, 48–55, 1981.

M.A. Schuette, J.P. Shen, D.P. Siewiorek, and Y.X. Zhu. Experimental evaluation of two concurrent error detection schemes. In *Digest of FTCS–16*, pages 138–143, IEEE, 1986.

R.K. Scott, J.W. Gault, and D.F. McAllister. Fault tolerant software reliability modeling. *IEEE Transactions on Software Engineering*, SE-13(5):582–592, 1987.

K.G. Shin and Y.-H. Lee. *Measurement of Fault Latency: Methodology and Experimental Results.* Technical Report CRL-TR-45-84, Computing Research Laboratory, University of Michigan, 1984.

S.K. Shrivastava, G.N. Dixon, and G.D. Parrington. *Objects and Actions in Reliable Distributed Systems.* Technical Report 242, Computing Laboratory, University of Newcastle upon Tyne, 1987.

T.B. Smith. High performance fault tolerant real time computer architecture. In *Digest of FTCS–16*, pages 14–19, IEEE, 1986.

R.F. Stone. *Reliable Computing Systems – A Review.* Technical Report YCS 110, Department of Computer Science, University of York, 1989.

A.N. Tantawi and M. Ruschitzka. Performance analysis of checkpointing strategies. *ACM Transactions on Computer Systems*, 2(2):123–144, 1984.

D.J. Taylor and M.L. Wright. Backward error recovery in a UNIX environment. In *Digest of FTCS–16*, pages 118–123, IEEE, 1986.

M. Thomas. Development methods for trusted computer systems. *Formal Aspects of Computing*, 1(1):5–18, 1989.

C.-L Yang and G.M. Masson. An efficient algorithm for multiprocessor fault diagnosis using the comparison approach. In *Digest of FTCS–16*, pages 238–243, IEEE, 1986.

J.W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.